Locking timestamps versus locking objects

Marcos K. Aguilera
VMware Research

Tudor David*
IBM Research, Zurich

Rachid Guerraoui
EPFL

Junxiong Wang
EPFL

ABSTRACT

We present *multiversion timestamp locking (MVTL)*, a new genre of multiversion concurrency control algorithms for serializable transactions. The key idea behind MVTL is simple: lock individual time points instead of locking objects or versions. After presenting a generic MVTL algorithm, we demonstrate MVTL's expressiveness: we give several simple MVTL algorithms that address limitations of current multiversion schemes, by committing transactions that previous schemes would abort, by avoiding the problem of serial aborts and ghost aborts, and by offering a way to prioritize transactions that should not be aborted. We give evidence that, in practice, MVTL-based algorithms can outperform alternative concurrency control schemes.

1 INTRODUCTION

Serializable transactions are a powerful paradigm available in many computing systems, such as transactional memory, database systems, and key-value storage systems. To ensure serializability, transactions require a scheme for concurrency control to handle any negative consequences of transaction interleaving.

The literature on concurrency control is rich [5, 30], and a particularly appealing class of algorithms is called *multiversion concurrency control* [4]. Briefly, these algorithms keep a *history* of each object, containing many versions of the data with associated timestamps. This history gives the system a *choice* of which version to use when an object is accessed. This choice permits more transactions to execute concurrently without blocking or aborting. For example, in some multiversion algorithms [5, 8], read-only transactions can execute without ever blocking or aborting, and update transactions can concurrently update the same object. Enabling more concurrency has become particularly important with the proliferation of multi-core and large-scale systems. Multiversion algorithms have wide application: they are used often in database systems both commercial and academic [10, 22, 30, 31], and more recent work has applied them to key-value storage systems and transactional memory (e.g., [11, 17–19, 25, 26]). In this paper, we do not restrict ourselves to particular applications, but rather study multiversion algorithms in their broadest scope.

There are three main genres of multiversion algorithms: *lock based*, *timestamp ordering*, and *serialization graph based* [5]. Lock-based algorithms (e.g., MV2PL [5]) acquire locks to avoid the ill-effects of concurrency; these algorithms are very simple. Timestamp ordering algorithms (e.g., MVTO [5]) assign a timestamp to each transaction, and then serialize transactions by timestamp; these algorithms permit read-only transactions to execute without ever aborting. Serialization graph algorithms (e.g., MVSGT [30]) detect cycles in the serialization graph to prevent a violation of serializability; these algorithms permit higher levels of concurrency than the alternatives.

*Work done while the author was at VMware Research and EPFL.

Despite their many benefits, all types of multiversion algorithms have limitations. Lock-based algorithms significantly restrict the degree of concurrency. Timestamp ordering algorithms are susceptible to aborts, including *serial aborts*—aborts in serial executions—and *ghost aborts*—aborts caused by a conflict with a transaction that already aborted. Serialization graph algorithms are complex and incur significant computation overheads [2, 18, 23].

In this paper, we introduce a new genre of multiversion algorithms, called *multiversion timestamp locking* or MVTL. MVTL is based on a simple idea: use locks as in lock-based algorithms, but lock individual timestamps of objects, rather than entire objects at a time. A transaction is allowed to commit if it can find at least one timestamp that it managed to lock across all its objects. Intuitively, MVTL performs well because it uses locks with fine granularity: not only individual objects have separate locks, but individual timestamps within objects have their own locks. Locking at fine granularity increases parallelism and decreases blocking and aborting, as the system can explore many serialization points for each transaction.

Conceptually, MVTL keeps a lock state for each object and each timestamp, which amounts to an infinitely large lock state. However, in practice we can reduce the lock state significantly using interval compression, so that each object holds just a few lock intervals, and this state can be subsequently discarded when the associated versions are purged.

To precisely define MVTL, we give a generic algorithm (§4) that has several nondeterministic choices, such as what timestamps each operation tries to lock, and how locks are acquired (wait or give up on blocked locks). We prove that these choices do not affect safety: the generic algorithm is correct irrespective of them. However, the choices are crucial for performance.

We then propose several specific algorithms that specialize the generic MVTL algorithm by fixing these choices to obtain different benefits (§5). These algorithms are simple and address some important drawbacks of existing multiversion algorithms, such as serial aborts, ghost aborts, the lack of a priority scheme for transactions, and more. We also show that pessimistic and timestamp ordering algorithms can be seen as special cases of MVTL. Thus, in a precise sense, MVTL unifies these algorithms.

Next, we discuss some practical considerations around MVTL, such as how to compress the lock state (§6). We separate out these considerations because they are orthogonal to the concepts underlying the MVTL algorithm. However, they are important to using MVTL in practice.

Then, we show how to extend the basic MVTL algorithm to distributed transactions in a message-passing system (§H). We believe MVTL is particularly relevant in this setting as it can be quite communication efficient.

We implement an MVTL-based algorithm, and compare its behavior with multiversion and lock-based alternatives. The results indicate significant advantages of MVTL in read-write workloads, and no disadvantages under read-only workloads.

To summarize, the contributions of this paper are as follows:

- We propose a new genre of multiversion algorithms for transactions, called multiversion timestamp locking (MVTL), which is based on the idea of locking timestamps.
- We give several MVTL algorithms, which address various limitations of current multiversion algorithms.
- We show that MVTL generalizes both multiversion timestamp ordering and pessimistic multiversion algorithms.
- We discuss practical considerations for implementing MVTL, including techniques to compress the lock state.
- We describe a version of MVTL for distributed transactions.
- We implement an MVTL-based algorithm and showcase its advantages over alternatives.

The main contribution is conceptual in nature: locking individual timestamps is a new way to approach multiversion algorithms. The specific MVTL algorithms we present are simple and just scratch the surface; the investigation of additional MVTL algorithms is an exciting direction for future work. Also interesting is to implement MVTL in other types of transactional systems, such as software transactional memory, transactional key-value storage systems, transaction object systems, and database systems. While the fundamental MVTL algorithms we present are system agnostic, the details of how these algorithms can be best implemented depend on the system and deserve further study.

For ease of exposition, some details of our contribution are given in the appendix, including proofs and pseudo-code of some algorithms.

2 MODEL

We consider a standard model for a multi-threaded concurrent system [14]. The system has processes that communicate via atomic shared memory. The system is asynchronous: there are no bounds on the relative speed of processes. We assume the existence of a discrete global clock with domain $\mathcal{T} = \{0, 1, \ldots\}$, and processes may or may not have access to the global clock. More precisely, processes may have local clocks that match the global clock ("synchronized clocks") or that are within a known bound ϵ of the global clock ("ϵ-synchronized clocks").

We are interested in algorithms that implement a transactional storage system. Such a system maintains a set of objects and allows processes to manipulate the objects using transactions. Each object has a unique key (identifier) and, by abuse of language, we refer to the object and its key interchangeably. The system supports four operations with their usual semantics: BEGIN(tx) starts a transaction tx, COMMIT(tx) tries to commit tx and returns a success indication, READ(tx, k) reads key k within tx, and WRITE(tx, k, v) writes v to k within tx. Transactions are dynamic: their read and write operations can depend on the results of prior operations in the transaction.

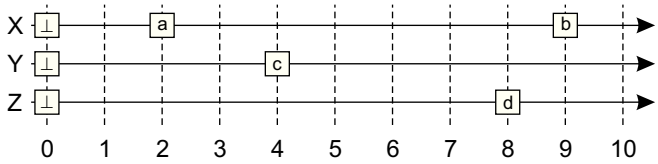
Our correctness condition is *multiversion view serializability*, a form of serializability well-suited for multiversion algorithms. Roughly speaking, this condition requires every multiversion schedule of the algorithm to be equivalent to a serial monoversion schedule [5, 30].

Some of our results refer to a *workload*, which specify the transactional work submitted to the system. More precisely, a workload is a sequence of operations indexed by the transaction they belong to, where each operation is *read*(k), *write*(k, v), or *commit*. We use workloads to study how different protocols react to the same inputs.

3 OVERVIEW

After recalling multiversion concurrency control, we introduce *timestamp locking* and explain how it addresses weaknesses of existing multiversion algorithms.

Multiversion concurrency control and the MVTO+ algorithm. The basic idea of multiversion timestamp ordering is to assign a timestamp to each transaction and then use the timestamp to determine (a) what version the transaction reads from, (b) what version it writes to, and (c) the serialization order of transactions. This idea can lead to several slightly different algorithms. To focus the discussion, here we present a concrete algorithm denoted MVTO+, which is identical to the MVTO algorithm in [5] but with an improvement: it avoids cascading aborts by not reading uncommitted data. For each object, MVTO+ keeps many versions and a timestamp for each version. It is useful to think of each object as an evolving timeline with values. Each transaction tx has a unique timestamp t, which determines the version of objects that tx reads and writes. Specifically, when tx reads an object, it obtains the version of the object with the largest timestamp before t. When tx writes an object, tx does not immediately produce a new version but instead it stores the written value in a temporary area for the transaction. Upon commit, tx takes each written value in this temporary area and produces a new version with timestamp t.



For example, the figure above depicts three objects X, Y, and Z. Each object has an initial version denoted $\perp$. In addition, X has two other versions with data a and b and timestamps 2 and 9; Y has data c with timestamp 4; and Z has data d with timestamp 8. Suppose a transaction tx is assigned timestamp 6. If tx reads X, it obtains a—the largest version with a timestamp before 6. Similarly, if tx reads Y, it obtains c. If tx writes e to Z and commits, then Z gets a new version with data e and timestamp 6.

Ultimately, transactions are serialized by the order of their timestamps. A key implication is that, after tx reads X and obtains a, another transaction should not produce a version of X with a timestamp between 2 and 6. To prevent this behavior, MVTO+ keeps a *read-timestamp* for each version: this is the largest timestamp with which the version was read by a transaction. In the example, after tx reads X and obtains a, the read-timestamp of a becomes 6 (if it was not already larger than 6).

Timestamp locking. We look at MVTO+ slightly differently, using our new notion of *timestamp locking*. This notion allows us to generalize MVTO+ into our new MVTL algorithm. Rather than read-timestamps, we can think that each object has several locks, one for each timestamp. When tx reads X, rather than updating the read-timestamp of a to 6, we can think that tx obtains a read-lock on each timestamp between 3 and 6. When another transaction

wishes to write a version with timestamp, say 5, it must obtain the write-lock on that timestamp. But the read-locks by *tx* prevent this from happening, as required by MVTO+. We can now see the read-timestamp of *a* as simply a compact representation of the fact that there are read-locks between 3 and 6.

Thinking about timestamp locks has several advantages over read-timestamps. First, with read-timestamps, it is not clear what should happen if *tx* aborts: should the read-timestamp of *a* be updated to its previous value? But what is the previous value if several other transactions read *a* concurrently? This is a hard question, and MVTO+ avoids it altogether by taking an unnecessarily conservative approach: when *tx* aborts, it leaves the read-timestamp of *a* at 6. We show that this choice leads to ghost aborts. In contrast, timestamp locks provide a better alternative: if *tx* aborts, its read-locks are removed but the read-locks of other transactions remain.

Second, with timestamp locks, there is no reason that a transaction should be restricted to obtaining write-locks on just one timestamp, or obtaining read-locks on a range that ends with the transaction's timestamp. Permitting more choices allows the system to avoid serial aborts, as we explain later.

These advantages are captured by our MVTL algorithm, which we now briefly summarize. With MVTL, when a transaction wishes to read an object, it selects a version of the object to read and obtains read-locks on one or more timestamps adjacent to and immediately following that version. To write an object, the transaction obtains write-locks on one or more timestamps anywhere. To commit, the transaction must find a single common timestamp that is read-locked or write-locked across all objects read or written by the transaction, respectively. If such a timestamp exists, the transaction commits; otherwise, it aborts.

The exact timestamps that are locked by reads and writes depend on a *locking policy*. The algorithm remains correct for any locking policy, but a poorly chosen policy causes many aborts because there is no common locked timestamp. We present some simple but interesting algorithms using various locking policies, each with its own advantages.

4 GENERIC MVTL ALGORITHM

We now present our generic MVTL algorithm in detail. We start with some basic concepts (§4.1), explain a simple lock extension we use (§4.2), and cover the main algorithm (§4.3). We present a centralized version of MVTL designed for a single server. We later describe a distributed version of MVTL intended for distributed transactions. Some practical considerations for implementing MVTL, including how locks and data can be compacted, are discussed in §6.

4.1 Preamble

The system keeps many versions of data in an array $Values[k, t]$ where k is a key and t is a timestamp. To ensure processes pick distinct timestamps, we add a process id to a timestamp; thus, a timestamp is a pair (v, p) ordered lexicographically, where v is a real number. There is a smallest timestamp denoted 0, and a special value denoted $\perp$, such that initially $Values[k, 0] = \perp$ for every k.

4.2 Freezable locks

The MVTL algorithm deals with *write-once objects*—objects initially set to $\perp$ that may change their state at most once. We define a simple variation of readers-writer locks, which we call freezable locks, which are appropriate for such objects and we use them in MVTL. A freezable lock is similar to a readers-writer lock, except that a lock holder can freeze the lock to indicate that it will never release it. Freezing is useful because it tells other processes that they should not wait to acquire the lock; we use this feature in several specialized MVTL algorithms. If a lock holder does not freeze a lock, it is expected to release it eventually.

We apply freezable locks to write-once objects as follows. A process acquires the lock in write mode if it intends to write the object. The process may ultimately fail to write if the transaction aborts, in which case it releases the lock; but if the transaction commits, the process freezes its lock to ensure other processes will not try to write the object again. Similarly, a process acquires the lock in read mode to read the object and it freezes the lock in case of a commit; if the object was not written (its state is $\perp$), this prevents other processes from writing to it, sealing its fate.

4.3 Algorithm

Algorithm 1 The generic MVTL algorithm (part 1/2): main code

```
1:  function BEGIN(tx)
2:      tx.readset ← ∅; tx.writeset ← ∅; tx.committs ← ⊥
3:  function WRITE(tx, k, v)                          ▷ write v to k in transaction tx
4:      WRITE-LOCKS(tx, k)                  ▷ write lock some subset of timestamps
5:      add (k, v) to tx.writeset           ▷ remember key and value we wrote
6:  function READ(tx, k)                              ▷ read k in transaction tx
7:      tr ← READ-LOCKS(tx, k) ▷ read lock some interval [tr+1, . . .] with
                                              Values[k, tr] ≠ ⊥
8:      if tr = ⊥ then return ⊥                                  ▷ read failed
9:      add (k, tr) to tx.readset           ▷ remember key and version we read
10:     return Values[k, tr]                              ▷ return committed value
11: function COMMIT(tx)                          ▷ try to commit transaction tx
12:     COMMIT-LOCKS(tx)                    ▷ locks to acquire at commit time
13:     T ← {t : ∀k ∈ tx.readset.keys, tx has a lock on (k, t) and
                ∀k ∈ tx.writeset.keys, tx has a write-lock on (k, t)}
                                  ▷ try to find a locked timestamp for tx
14:     if T = ∅ then mark tx as aborted
15:     else
16:         tx.committs ← COMMIT-TS(T)          ▷ pick some timestamp in T
17:         for (k, v) ∈ tx.writeset do
18:             freeze write-lock for tx on (k, tx.committs)   ▷ freeze locks
19:             Values[k, tx.committs] ← v          ▷ expose committed value
20:         mark tx as committed
21:     if COMMIT-GC(tx) then GC(tx)                          ▷ invoke gc or not
22: function GC(tx)                ▷ garbage collect locks of tx after it ended
23:     if tx committed then
24:         for (k, tr) ∈ tx.readset do
25:             freeze read-locks for tx on [tr+1, tx.committs]
26:     release all unfrozen read- and write-locks for tx
```

Algorithm 1 shows the main code of the generic MVTL algorithm. For clarity, we assume that the code in lines 17–19 is executed atomically, but we later remove this assumption (§6). To write a value

into key k, a transaction obtains zero or more write-locks on timestamps for that key (function WRITE-LOCKS in line 4). Intuitively, a write-lock on a timestamp t for key k allows the transaction to commit with timestamp t as far as accesses to k are concerned. After getting the locks, the transaction remembers the key and value; the write is not visible to other transactions until the transaction commits.

To read a key, a transaction gets zero or more read-locks on timestamps for that key (function READ-LOCKS in line 7), with the requirement that these timestamps form a contiguous interval that starts immediately after the version that the read returns. For instance, if $[tr{+}1, te]$ denotes the read-locked timestamps, then the read must return the value committed with timestamp tr. This requirement is necessary for serializability: intuitively, the read locks permit the transaction to commit with any timestamp $t \in [tr{+}1, te]$ after having read v, by preventing other transactions from writing a different value with a timestamp between tr and te. After locking, the transaction remembers k and tr; knowledge of k is necessary to commit, and knowledge of both k and tr is needed to garbage collect the locks of the transaction.

To commit, a transaction gets zero or more additional locks (function COMMIT-LOCKS in line 12) and tries to find a commit timestamp t that is write-locked for every k in the write-set, and that is read- or write-locked for every k in the read-set. (A key in the read-set may be write-locked because the transaction read the key and then wrote it.) If there are many such timestamps, the transaction picks one (function COMMIT-TS in line 16). The transaction then freezes write-locks on that timestamp and records the written values so that they can be seen by other transactions. As an optional step (as determined by calling COMMIT-GC in line 21), the transaction may garbage collect the locks it holds. Doing so freezes the read locks between the version read and the commit timestamp, and releases all other locks. If the algorithm skips garbage collection on commit, garbage collection can be invoked any time later in the background; this is not shown in the code.

Algorithm 2 The generic MVTL algorithm (part 2/2): policy

```
1: function WRITE-LOCKS(tx, k)
2:     acquire write-locks for tx on (k, T) for some set T
3: function READ-LOCKS(tx, k)                    ▷ returns a timestamp or ⊥
4:     acquire read-locks for tx on (k, T) for some T = [tr+1, . . .] where
                Values[k, tr] ≠ ⊥
5:     either return tr or return ⊥
6: function COMMIT-LOCKS(tx)
7:     acquire read- or write-locks for tx on some keys and timestamps
8: function COMMIT-TS(T) return some t ∈ T
9: function COMMIT-GC(tx) either return true or return false
```

The algorithm depends on a policy of what locks to acquire, how to pick one of many possible commit timestamps, and whether to garbage collect during commit; these choices can depend on the transaction and other considerations. The choices are determined by the functions that we mentioned above: WRITE-LOCKS, READ-LOCKS, COMMIT-LOCKS, COMMIT-TS, and COMMIT-GC. The generic MVTL algorithm uses a generic policy that makes these choices nondeterministically (Algorithm 2). For example, to obtain write locks, the generic policy nondeterministically picks a set T of timestamps to lock. To obtain read locks, the policy picks an interval of timestamps starting immediately after a committed version.

We prove that the generic MVTL algorithm is correct with its nondeterministic choices (see Appendix A). Naturally, this correctness carries over to any specialization that fixes the nondeterministic choices. These specializations lead to different algorithms (§5).

Some policies of the generic algorithm may cause deadlocks, where a process waits forever to acquire a lock. In such cases, standard techniques for deadlock detection can be used to abort the required transactions (e.g., cycle detection in the wait-for graph, timeout, etc). In Appendix A, we show the following:

THEOREM 1. *The generic MVTL algorithm (Algorithms 1 and 2) ensures serializability.*

5 SIMPLE MVTL ALGORITHMS

We now give several simple algorithms that are special cases of the generic MVTL algorithm, each with a different benefit. To specify these algorithms, we specialize the generic policy of MVTL (Algorithm 2). We provide proofs and pseudo-code substantiating our claims regarding these applications in the appendices.

5.1 The preferential algorithm

Roughly speaking, the preferential algorithm, denoted MVTL-Pref, works with multiple timestamps for each transaction, where one of the timestamps is preferential. The algorithm tries to commit a transaction using its preferential timestamp, but if doing so would abort, it tries one of the other timestamps. To ensure viability of the other timestamps, the algorithm locks them as necessary during the execution.

More precisely, MVTL-Pref is parameterized by a function $A(t)$ that takes the transaction's preferential timestamp and returns a non-empty set of alternative timestamps different from t. $A(t)$ is a choice of the user of the algorithm. For example, $A(t) = \{t{-}10, t{+}10\}$ indicates that $t{-}10$ and $t{+}10$ are the alternative timestamps for a transaction with preferential timestamp t. The preferential timestamp itself comes from a clock, as in other timestamp-based protocols.

We assume that clock timestamps are unique (e.g., by appending the process id to each timestamp t) and that $A(t)$ also produces unique timestamps (e.g., by using the process id in t for each timestamp in $A(t)$).

When executing a read on a key k, the algorithm determines a version to return based on the preferential timestamp, and then read-locks contiguous timestamps of k to cover as many alternative timestamps as possible. When executing a write to key k, the algorithm obtains no locks; rather, locks are acquired at commit time, as follows. If the algorithm cannot obtain a write-lock for the preferential timestamp for each written key, it tries one of the alternative timestamps. If it manages to obtain read- and write-locks for all read and written objects at one of the timestamps, the transaction commits; otherwise it aborts.

We can show that if we choose the alternative timestamps $A(t)$ to be smaller than the preferential timestamps t, then the resulting MVTL-Pref algorithm aborts strictly fewer workloads compared to MVTO+. More precisely:

Algorithm 3 The MVTL-Pref algorithm

```
1: function Initialization(tx)
2:     tx.PrefTS ← clock()
3:     tx.PossTS ← {tx.PrefTS} ∪ A(tx.PrefTS)
                                        ▷ possible timestamps for tx
4: function write-locks(tx, k) return ▷ lock write-set only on commit
5: function read-locks(tx, k)
6:     repeat
7:         tr ← max{t : t < tx.PrefTS and Values[k, t] ≠ ⊥}
                                        ▷ candidate value to read
8:         tmax ← max{t ∈ tx.PossTS :
                        no timestamps in [tr+1, tmax] are write frozen}
9:         for t ← tr+1 to tmax do   ▷ read-lock [tr+1, tx.TS] if possible
10:            try to acquire read-lock for tx on (k, t), waiting
                   if timestamp is write-locked but not frozen
11:            if found frozen write-lock then
                   release read-locks acquired above; break ▷ exit "for" loop
12:    until found no frozen locks in the for loop
13:    tx.PossTS ← tx.PossTS ∩ [tr, tmax]   ▷ update possible timestamps
14:    return tr
15: function commit-locks(tx)
16:    for t ∈ tx.PossTS do   ▷ Find a good timestamp. Loop order: first
                                  tx.PrefTS then arbitrary for PossTS
17:        gotlocks ← true
18:        for (k, tr) ∈ tx.writeset do
19:            try to write-lock for tx on (k, t), without waiting if a
                   timestamp is read-locked
20:            if write-lock not acquired then
21:                gotlocks ← false        ▷ this timestamp will not work
22:                release all write locks for tx
23:                break                   ▷ exit inner "for" loop
24:        if gotlocks then break ▷ found a timestamp for which we can
                                    get write locks; exit outer "for" loop
25:    if gotlocks then tx.TS ← t          ▷ found good timestamp
26:    else tx.TS ← ⊥                      ▷ no good timestamps
27: function commit-ts(T) return tx.TS
28: function commit-gc(tx) return false
```

Theorem 2. *Suppose that $\forall t' \in A(t), t' < t$. (a) If a workload W produces no abort under MVTO+, then W produces no abort under MVTL-Pref. (b) There are infinitely many workloads that produce no aborts under MVTL-Pref but produce aborts under MVTO+.*

The pseudo-code of MVTL-Pref is given in Algorithm 3.

5.2 The prioritizer algorithm

Multiversion timestamp ordering provides no way for critical transactions to be prioritized over normal transactions. We explain how MVTL can do that, by using a policy that gives more locks to critical transactions. There are many ways to do that, but the simplest one is as follows. Normal transactions obtain their locks as in multiversion timestamp ordering using synchronized clocks, while critical transactions try to acquire all locks as in pessimistic concurrency control except that critical transactions do not block waiting for any of its locks. Both types of transactions garbage collect on commit.

Theorem 3. *In the MVTL-Prio algorithm, transactions labeled critical are never aborted by transactions labeled normal.*

Given that high-priority transactions behave similarly to pessimistic concurrency control, they can cause deadlocks. However, transactions with normal priority behave identically to those in MVTO+, and thus never cause deadlocks.

5.3 The ϵ-clock algorithm

Multiversion timestamp ordering uses clocks to obtain its timestamps, but if clocks are not synchronized or monotonic[1], it is susceptible to *serial aborts*—aborts that occur in an execution that is completely serial. This is a concern in modern multicore machines that do not guarantee that clocks across cores are perfectly synchronized. For example, T_2 gets timestamp 2, reads an object X, and commits. Afterwards, T_1 gets a smaller timestamp 1, writes X, and tries to commit. This will cause T_1 to abort since the read-timestamp of X at version 0 is 2. This is the schedule:

$$\begin{array}{lcccc} T_2: & R(X) & C & & \\ T_1: & & & W(X) & A \end{array}$$

Here, time flows to the right and each line shows the operations of a transaction. R, W, C, and A indicate a read, write, commit, and abort; and X is the key. Thus, this schedule has two transactions T_1 and T_2, where T_2 reads X and commits, and then T_1 writes X and aborts.

The MVTL-ϵ-clock algorithm, which we now introduce, avoids serial aborts when used with ϵ-synchronized clocks. Briefly, when it starts, a transaction reads the clock, obtains a time t, and for each read and write tries to lock the interval $[t-\epsilon, t+\epsilon]$. At the end, it commits at the smallest common timestamp it locked for every accessed object. Before completing the commit, the transaction runs garbage collection. In a sequential execution, it is possible to show that tx picks a commit timestamp that is at most t, and thus it releases the lock on higher timestamps. As a result, the next transaction in the sequence will always have its own real time in the intersection of locked time points, and therefore does not abort. Alforithm 4 shows the pseudo-code of MVTL-ϵ-clock.

Theorem 4. *The MVTL-ϵ-clock algorithm is not susceptible to serial aborts when clocks are ϵ-synchronized.*

5.4 Existing algorithms as special cases

We now show that MVTL generalizes two popular transactional algorithms, MVTO+ and pessimistic concurrency control. More precisely, we give two algorithms MVTL-TO and MVTL-Pessimistic, which specialize MVTL and behave exactly like MVTO+ and pessimistic concurrency control, respectively

In MVTL-TO, each transaction obtains a timestamp t from a clock when the transaction starts. Writes do not lock anything, reads try to lock $[tr+1, t]$ (waiting for unfrozen locks) where tr is the largest timestamp before t for which $Values[k, tr] \neq \bot$, and commits lock t for each object in the transaction's write-set. Garbage collection is not invoked on commit.

Theorem 5. *The MVTL-TO algorithm behaves as the MVTO+ algorithm.*

[1] A monotonic clock is one that ensures that it returns a higher timestamp if it is queried later in time. Monotonic clocks and time-synchronized clocks are equivalent insofar this discussion is concerned.

Algorithm 4 The MVTL-ϵ-clock algorithm

```
1:  function Initialization(tx)
2:      now ← clock()
3:      tx.TS ← [now − ε, now + ε]
4:  function write-locks(tx, k)
5:      try to write-locks for tx on (k, tx.TS), waiting
            if a timestamp is read- or write-locked but not frozen
6:      tx.TS ← write-locks that tx could acquire
7:  function read-locks(tx, k)
8:      if tx.TS = ∅ then return ⊥
9:      m ← max tx.TS
10:     repeat
11:         tr ← max{t : t < m and Values[k, t] ≠ ⊥}
12:         for t = tr+1 to m do ▷ read-lock interval [tr+1, m] if possible
13:             try to acquire read-lock for tx on (k, t), waiting
                    if timestamp is write-locked but not frozen
14:             if found frozen write-lock then
                    release read-locks acquired above; break ▷ exit "for" loop
15:     until found no frozen locks in the for loop
16:     tx.TS ← tx.TS ∩ [tr+1, m]
17:     return tr
18: function commit-locks(tx) return
19: function commit-ts(T) return min T
20: function commit-gc(tx) return true
```

Pessimistic concurrency control locks objects before accessing them, to prevent conflicting operations from executing concurrently. To emulate pessimistic concurrency control using MVTL, writes acquire write locks on all timestamps (blocking), while reads acquire read-locks on all timestamps in $[tr{+}1, \infty]$ (blocking). Garbage collection is invoked on commit.

Theorem 6. *The MVTL-Pessimistic algorithm behaves as the pessimistic concurrency control algorithm.*

5.5 The ghostbuster algorithm

Under multiversion timestamp ordering, a transaction may abort and later create a conflict with another transaction, causing it to abort. For example, suppose that T_1 starts with timestamp 1, T_2 starts with timestamp 2, and T_3 starts with timestamp 3. Then T_3 reads X and commits, T_2 reads Y, writes X, and tries to commit with its timestamp 2, but T_2 aborts because T_3 read X with timestamp 3. Next T_1 writes Y and tries to commit but aborts due to the read by T_2. This is a ghost abort, because the write of T_1 has a conflict with a transaction T_2 that had aborted before the write of T_1 started. This is the schedule:[2]

$$
\begin{array}{lccccccc}
T_3: & R(X) & C & & & & & \\
T_2: & & & R(Y) & W(X) & A & & \\
T_1: & & & & & & W(Y) & A
\end{array}
$$

We define ghost aborts precisely in Appendix G.

While multiversion timestamp ordering has ghost aborts, MVTL-Ghostbuster can avoid that. MVTL-Ghostbuster is a simple modification to the MVTL-TO algorithm (§5.4): when a transaction commits, it performs garbage collection. This ensures that transactions that abort do not leave behind locks that cause ghost aborts. We thus claim the following:

Theorem 7. *The MVTL-Ghostbuster algorithm is not susceptible to ghost aborts.*

6 PRACTICAL CONSIDERATIONS

Reducing lock state space. When we presented the generic MVTL algorithm, we defined a lock for each timestamp and object, which amounts to an infinite lock state space. We did not include mechanisms to compress this information, because they are orthogonal to the essence of the algorithm. However, a practical implementation should compress the lock state space. To do so, we observe that MVTL algorithms usually acquire and release locks on a small number of points or contiguous intervals (this is true for all algorithms we presented).Rather than keeping a lock state for each timestamp, an implementation can keep a single lock state for an entire interval. In the algorithms we presented, each object holds at most one lock interval per committed transaction. We evaluate the amount of lock state required in §8.4.5. Furthermore, this state can be discarded when the associated version of the object is purged, as we discuss next.

Purging versions. By nature, a multiversion algorithm keeps multiple versions of each object. Doing so is feasible as storage prices fall. Disk systems such as database systems already use multiversion algorithms, but even memory systems are targets now. Nevertheless, multiversion algorithms need a way to purge old versions so that each object holds few versions—possibly just one after write activity on the object quiesces. We now explain how this can be done in MVTL. This is easy: at any time, the system can purge any version older than the latest committed one, without affecting the correctness of the algorithm. Transactions that need purged versions will abort, so in practice we purge versions older than a time limit chosen based on the duration of its longest transactions. In some MVTL algorithms, there is a lower bound on the timestamps that a transaction locks (e.g., ϵ-clock algorithm); we can purge versions with timestamps below the bound except the last one before the bound, without causing any side-effects. We evaluate the effectiveness and cost of garbage collection in §8.4.5.

Removing the atomic block. Algorithm 1 has an atomic block in lines 17–19, to avoid partially exposing the writes of a committing transaction when we assign to the array $Values[k, t]$. We can remove this atomic block by (1) first storing a special value in $Values[k, t]$ for all timestamps in the for loop, (2) then storing the actual value v for all timestamps in the loop, and (3) having other processes wait if they read *Values* and see the special value.

7 DISTRIBUTED MVTL ALGORITHM

We now explain how to extend the generic MVTL algorithm of §4 to distributed transactions. The system consists of a set of clients who want to execute transactions, and a set of storage servers who keep the data, where clients and servers are connected by a network. The data is partitioned across the servers by its key, and clients

[2] Here, transactions get a timestamp before their first operation, but one can construct a more complex schedule with the same problem even if transactions get a timestamp at the first operation.

know how to find the server responsible for a key (e.g., by hashing the key or using a configuration map).

The basic idea of the distributed algorithm is that servers hold the state that is shared across clients: locks and data versions. Clients contact the servers to execute the steps of the algorithm in §4 that involve this state. More precisely, the server responsible for a key k keeps all versions and locks for k. A client contacts that server when it wishes to read k, create a new version for k, or manipulate k's lock state (obtain, freeze, or release locks on timestamps).

The system is subject to failures that may disrupt the system. A failed client may leave write locks in an unfrozen state indefinitely, causing other transactions to block forever. A failed server can similarly cause either indefinite waiting from clients.

To address these problems, we associate a *commitment* object with each transaction. This object solves consensus on the outcome of a transaction, which can be "abort" or "commit with a timestamp t", ensuring that clients and servers all agree on the outcome. The details of the algorithm are given in the Appendix H.

8 EXPERIMENTAL EVALUATION

We conduct a simple experimental evaluation of MVTL to answer some questions: Does MVTL enhance transaction concurrency and avoid aborts compared to alternatives? Does MVTL improve transaction throughput? On which workloads? Do the characteristics of the environment impact our conclusions? Does MVTL incur significant overheads in terms of state size?

To this end, we implement the distributed MVTL algorithm (§H) with a variant of the ϵ-clock algorithm (§5.3). In this variant, to execute a transaction T, a client obtains a timestamp t from its local clock and associates a timestamp interval $I = [t, t + \Delta]$ with T, where Δ is a small constant (we pick $\Delta = 5$ms in the experiments). When accessing a key k, the client tries to lock the timestamps in I for key k. If the client cannot lock the entire interval I, but manages to lock some subinterval, then the client replaces I with that subinterval to reduce the amount of locking on subsequent keys. We call this algorithm MVTIL. This is similar to the ϵ-clock algorithm but we do not assume that clients have synchronized clocks and we shrink I when clients fail to obtain some locks, as described above. We consider two variants of MVTIL: (i) *MVTIL-early*, which at commit time picks the smallest timestamp in I to commit, and (ii) *MVTIL-late*, which picks the largest. We compare MVTIL to 2PL and MVTO+.

8.1 Implementation details

Keys and values are small strings of 8 characters. Clients are multi-threaded, each thread running a different transaction. When a client realizes that an ongoing transaction will abort (because it does not have a single timestamp locked across all accessed keys), it has the option of aborting or restarting the transaction, with an interval I adjusted based on the state it has already seen at the servers. Servers are multi-threaded, with hundreds of threads, each responsible to handle a client request. A server stores version and lock state in a hash table indexed by key; for each key, the hash table stores two skip lists, one for version state, one for lock state. The version state is a list of value-timestamp pairs ordered by timestamp. The lock state is a list of timestamp-timestamp pairs representing a locked time interval, ordered by the first timestamp. To coordinate access across threads, we use a concurrent hash table (from the Intel TBB library [16]), with a latch per entry in the hash table. Latches are held while a thread changes the lock and version lists of a key. We use Apache Thrift [1] for communication between clients and servers,

A timestamp service periodically broadcasts a message with a time T in the past, equal to the service's current time minus a constant K (we use $K = 15s$ in the local test bed, and $K = 60s$ in the cloud test bed – see below). This message has two effects. First, it causes servers to purge old versions of keys, namely versions that meet two criteria: their timestamp is smaller than T and they are not the most recent version of a key. If clients have ongoing transactions that later try to access a purged version, those transactions are aborted. However, because T is an old timestamp, there will be only few such transactions, if any. The second effect of broadcasting T is that clients advance their local clocks to T if they are behind—this ensures that clients with slow clocks do not start new transactions that need purged versions and subsequently get aborted.

Our implementations of MVTO+ and 2PL use the same framework, but run a different client protocol and keep a different server state: 2PL stores a single reader-writer lock per key, while MVTO+ stores a single skip list per key containing versions and associated locks. The implementations of all schemes are available at https://github.com/LPD-EPFL/MVTIL.

8.2 Test beds

We use two test beds for the experiments: a local test bed with dedicated servers and a public cloud test bed with virtual machine instances. The local test bed represents an enterprise setting with higher-performance machines and network, while the cloud test bed represents a low-cost shared environment with a less predictable network.

On the local testbed, we use three machines: (a) a server with four 2.7 GHz Intel Xeon 12-core E7-4830v3 processors and 512 GB of RAM; (b) a server with two 2.8 GHz Intel Xeon 10-core E5-2680 v2 processors and 256 GB of RAM; and (c) a server with four 2.1 GHz AMD Opteron 6172 12-core Processors and 128 GB of RAM. Machines are connected by a 1 Gbps network.

The public cloud test bed consists of several hundred Amazon EC2 *t2.micro* instances with 1 vCPU each.

8.3 General framework

In an experiment, clients submit transactions repeatedly in a closed-loop. We measure the aggregate throughput of committed transactions and the commit rate, which is the fraction of transactions that commit. Before measuring, we run a warm-up stage of 40s to ensure all clients have started; we then measure the system for 20s. We repeat each experiment five times and report the average.

In each experiment, we fix the following parameters:

- The algorithm (MVTIL, MVTO+, 2PL);
- The number of clients, which determine the level of concurrency;
- The size of transactions in number of operations;
- The fraction of write operations in a transaction;
- The size of the key space; and
- The number of storage servers.

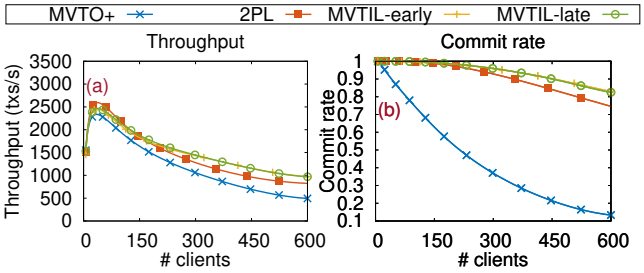


Figure 1: Effect of concurrency level on performance, local test bed.

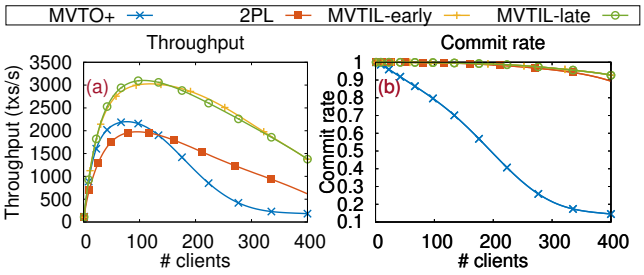


Figure 2: Effect of concurrency level on performance, cloud test bed.

On the local test bed, which has three machines, we always run three servers on all machines, and we run clients threads on a subset of the cores in those machines. For the cloud test bed, we run eight servers unless otherwise indicated, and we run each client on its own VM instance.

8.4 Results

We now present results regarding concurrency, fraction of write operations, transaction size, number of servers, and state size.

8.4.1 Level of concurrency. We study the effect of the level of concurrency on performance, under a workload where a majority of operations are reads—a common situation in practice. We vary the number of clients, while keeping the other parameters constant. We use transactions with 20 operations, 25% of which are writes. For the local test bed, we use 10K keys. For the larger cloud test bed, we use 50K keys.

Figures 1 and 2 show throughput and commit rates for the local and cloud test beds, respectively. We see that MVTIL outperforms MVTO+ and 2PL in both test beds. Moreover, when concurrency increases, the commit rate of MVTO+ drops due to conflicts, but this does not happen for MVTIL because it can commit at many serialization points. The inefficiency—due to aborts in MVTO+ and waiting for locks in 2PL—is the reason for the difference in throughout. This is more pronounced in the cloud test bed, where resources are scarce: there, MVTIL has roughly 2x better throughput than the alternatives. The difference is smaller on the local test bed.

The commit rate for 2PL is not optimal because we use timeouts: if a transaction makes no progress after a given time, we abort it. This prevents both deadlocks, and starving transactions from limiting throughput. In our experiments, we set the timeout such as to maximize total throughput.

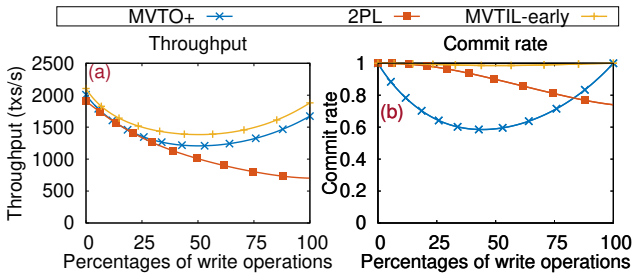


Figure 3: Effect of fraction of writes on performance.

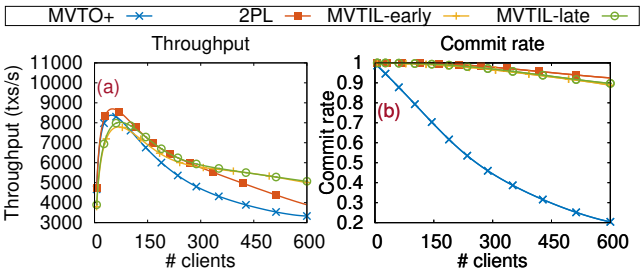


Figure 4: Effect of small transaction size on performance.

8.4.2 Write percentage. We next consider how the fraction of writes affect performance. We thus vary the fraction of writes and keep other parameters constant. We use the local test bed with 90 clients; transactions have 20 operations and 10K keys.

Figure 3 shows the result. For read-only transactions, the choice of protocol has little impact. Additionally, for write fractions close to 1, the workload consists mostly of blind writes, which allows multiversion protocols to commit nearly all transactions, as writes in such protocols do not conflict with each other. With a more balanced write fraction, MVTIL outperforms MVTO+ and 2PL. With 2PL, the more writes, the more time transactions wait for locks. MVTO+ has a high abort rate when the percentages of reads and writes are similar; this is where the chance of conflicts is highest in multiversion protocols. The issue impacts MVTIL less due to its ability to explore many serialization points to commit.

8.4.3 Transaction size. In previous experiments, we use transactions with 20 operations; we now consider smaller transactions with 8 operations. We vary the number of clients (level of concurrency) and observe the performance. We use the local test bed with a 50% fraction of writes and 10K keys.

Figure 4 shows the results. For a low concurrency, MVTIL behaves similar to MVTO+ and 2PL, but 2PL is ≈5% faster. This setting with little concurrency, short transactions, and a local test bed with lots of resources is the only setting where MVTIL is worse than an alternative. However, as we increase concurrency, MVTIL again outperforms the others. This advantage is larger in the cloud test bed (not shown).

8.4.4 Number of servers. We now consider how the number of servers affect performance. Using the cloud test bed, we keep the number of clients constant to 400 and vary the number of server instances from 1 to 20. We use transactions with 20 operations with 25% or 50% writes, and 100K keys.

Figure 5 shows the result. The throughput of all protocols increases with the number of servers, but the scalability is better for

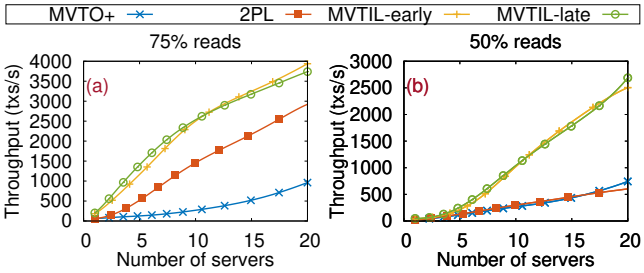


Figure 5: Effect of number of servers on performance.

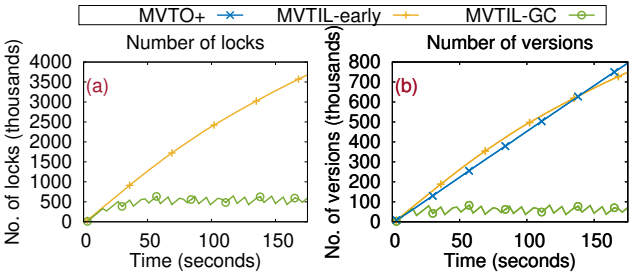


Figure 6: Number of locks and versions as time passes with garbage collection on and off.

MVTIL. MVTIL has a higher commit rate compared to MVTO+, and waits less for locks compared to 2PL; this is particularly visible with 50% writes.

8.4.5 State size. We now examine the size of the state kept by each algorithm, and the effectiveness of the garbage collection mechanisms. Most of the state of a multiversion protocol is the data versions and associated locks. We measure how the number of versions and locks evolve with time for MVTIL and MVTO+ without garbage collection, as well as MVTIL with garbage collection (MVTIL-GC) that activates every 15s to purge versions and locks. We use 50 clients running transactions with 20 operations, a fraction of 50% writes, and 8K keys, running on the local test bed.

Figure 6 shows the results. Without metadata purging, the state increases linearly with time. However, with garbage collection, the state size remains bounded in both the number of versions and locks (on average, ≈4 versions and ≈20 locks per key). Figure 7 shows how performance varies as time passes. Without garbage collection, throughput decreases after ≈5 minutes for MVTIL and MVTO+, because a larger state makes it slower to search for and access versions. Garbage collection removes this performance degradation. Moreover, comparing the performance with and without garbage collection at the beginning of the experiment, we see that the overhead of garbage collection is small.

8.5 Summary

We see that (i) with moderate contention, MVTIL outperforms alternatives, (ii) with no contention, MVTIL is at least as good as alternatives, and (iii) MVTIL's advantages are bigger in the cloud test bed that has limited processing power and unpredictable network latencies. MVTIL nevertheless represents just one of many MVTL-based algorithms. We believe that other MVTL algorithms will shine on other workloads and environments.

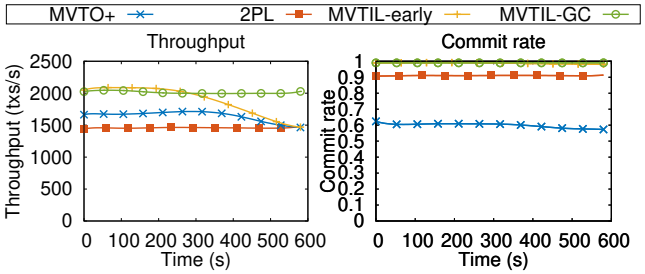


Figure 7: Performance as time passes with garbage collection on and off.

9 RELATED WORK

The main novelty of this work is the idea of locking individual timestamps, leading to a genre of multiversion algorithms called MVTL. No other work proposes this idea, but because MVTL is a broad class, several existing algorithms become special cases of MVTL, leading to similarities in mechanism.

Multiversion concurrency control is an old idea [5] that has seen a resurgence in software transactional memory (STM) systems, several of which provide serializability [2, 7, 11, 17, 18, 23, 25–27]. Prior work in this space falls into three categories: (1) multiversion for read-only transactions, (2) conflict graph schemes, and (3) multiversion timestamp ordering algorithms. The first category [11, 25–27] are systems that use multiversion to benefit solely read-only transactions; update transactions rely on optimistic methods that, upon commit, validate the read-set and abort if any object has changed. While read-only transactions are important, these methods abort under simple concurrent update schedules, such as the following (where full multiversion schemes do not abort):

$$\begin{array}{lccc} T_1: & R(X) & & W(Y) \\ T_2: & & W(X) & \end{array}$$

The second category (e.g., [2, 18, 23]) are multiversion systems that ensure serializability by detecting cycles in the *conflict graph*—a data structure that represents the conflicts across transactions—similarly to the MVSGT algorithm [30]. These algorithms have two drawbacks: they are complex and they incur significant computation overhead, as reported in some of these papers.

The third category [19] are systems that extend multiversion timestamp ordering. Specifically, Kumar et al. [19] explain how to provide opacity, which is stronger than serializability. However, the algorithm suffers from the same drawbacks of multiversion timestamp ordering that we address in §5. It should be possible to extend MVTL to provide opacity using the ideas of Kumar et al. [19], but this is future work.

Lomet et al. [22] introduce the multiversion timestamp range algorithm (MVTR). With MVTR, each transaction is assigned a range of timestamps, and this range shrinks as the transaction executes; at the end, MVTR commits if the range is non-empty. MVTR differs from MVTL because MVTR locks entire objects instead of timestamps. As a result, MVTR does not enjoy the full benefits of multiversion concurrency control, such as allowing two concurrent transactions to write the same object. Also, with MVTR one transaction manipulates the inner state of another transaction (e.g., by changing the range that another transaction uses), which requires careful synchronization of transactions using a scheduler or locks.

Elastic transactions [12], aimed at search data structures, use timestamp ranges to determine if a transaction can commit based on its start time and when the accessed objects were written.

Snapshot isolation [3] is both an isolation property and a protocol. The protocol uses multiversioning and timestamps, similarly to multiversion timestamp ordering, but it does not provide serializability. Other protocols that use multiversioning and timestamps provide even weaker notions than snapshot isolation [28].

Optimistic concurrency control (OCC) [20] is another technique that can use multiversioning. With OCC, a transaction does not acquire locks when executing; to commit, the system checks that the versions that the transaction read are the latest. TicToc [32] optimizes OCC to serialize transactions based on the data they access. TicToc computes potential serialization points before the validation and commit phases. Thus, a transaction for which the read and write sets have been inspected might later abort. In contrast, MVTL ensures that once a serialization point has been found, the transaction commits. Bohm [10] is a multiversion protocol that pre-orders transactions before execution; in that sense, Bohm is more pessimistic than MVTL, which determines transaction ordering dynamically during execution. In addition, Bohm requires that the transaction be known ahead of time, and that its write-set be static.

Many practical systems with distributed transactions provide only snapshot isolation [9, 24] and abort on concurrent writes to the same object. Spanner [8] provides strict serializability using two-phase locking for read-write transactions, which limits parallelism.

10 CONCLUSION

This paper introduced a new genre of multiversion concurrency control algorithms called multiversion timestamp locking (MVTL). MVTL offers a new way to look at multiversion algorithms, based on locking individual time points. With this perspective, we can find simple algorithms that improve the state of the art in various ways: by committing successfully more workloads than existing multiversion protocols, by avoiding the problems of serial aborts and ghost aborts, and by offering prioritized transactions. We can also view existing algorithms, such as MVTO and pessimistic concurrency control, as special cases of MVTL. Moreover, we showed how to realize MVTL in both centralized and distributed systems. Finally, we show experimental evidence of the benefits of MVTL in practice.

We believe that the algorithms proposed here are only a starting point for other possibilities opened up by MVTL. The design of other MVTL algorithms is a promising direction for future research.

Acknowledgements. We thank Dahlia Malkhi for valuable input early in the project. This work was supported in part by a VMware Fellowship.

REFERENCES

[1] Apache Thrift [n. d.]. https://thrift.apache.org
[2] Utku Aydonat and Tarek S. Abdelrahman. 2008. Serializability of transactions in software transactional memory. In *ACM SIGPLAN Workshop on Transactional Computing*.
[3] Hal Berenson, Phil Bernstein, Jim Gray, Jim Melton, Elizabeth O'Neil, and Patrick O'Neil. 1995. A critique of ANSI SQL Isolation Levels. In *International Conference on Management of Data*.
[4] Philip A. Bernstein and Nathan Goodman. 1981. Concurrency Control in Distributed Database Systems. *Comput. Surveys* 13, 2 (June 1981), 185–221.
[5] Philip A. Bernstein, Vassos Hadzilacos, and Nathan Goodman. 1987. *Concurrency control and recovery in database systems.*
[6] Christian Cachin, Rachid Guerraoui, and Luis Rodrigues. 2011. *Introduction to reliable and secure distributed programming.* Springer.
[7] João Cachopo and António Rito-Silva. 2006. Versioned Boxes As the Basis for Memory Transactions. *Science of Computer Programming* 63, 2 (2006), 172–185.
[8] J. Corbett, J. Dean, M. Epstein, A. Fikes, C. Frost, JJ Furman, S. Ghemawat, A. Gubarev, C. Heiser, P. Hochschild, W. Hsieh, S. Kanthak, E. Kogan, H. Li, A. Lloyd, S. Melnik, D. Mwaura, D. Nagle, S. Quinlan, R. Rao, L. Rolig, Y. Saito, M. Szymaniak, C. Taylor, R. Wang, and D. Woodford. 2012. Spanner: Google's Globally-Distributed Database. In *Symposium on Operating Systems Design and Implementation.*
[9] Jiaqing Du, Sameh Elnikety, and Willy Zwaenepoel. 2013. Clock-SI: Snapshot isolation for partitioned data stores using loosely synchronized clocks. In *IEEE Symposium on Reliable Distributed Systems.*
[10] Jose M Faleiro and Daniel J Abadi. 2015. Rethinking serializable multiversion concurrency control. *Proceedings of the VLDB Endowment* 8, 11 (July 2015), 1190–1201.
[11] Pascal Felber, Christof Fetzer, Patrick Marlier, and Torvald Riegel. 2010. Time-based software transactional memory. *IEEE Transactions on Parallel and Distributed Systems* 21, 12 (2010), 1793–1807.
[12] Pascal Felber, Vincent Gramoli, and Rachid Guerraoui. 2009. Elastic transactions. In *International Symposium on Distributed Computing.*
[13] Vassos Hadzilacos and Sam Toueg. 1994. *A modular approach to fault-tolerant broadcasts and related problems.* Technical Report TR 94-1425. Cornell University, Dept. of Computer Science, Cornell University, Ithaca, NY 14853.
[14] Maurice Herlihy and Nir Shavit. 2012. *The Art of Multiprocessor Programming, Revised First Edition.*
[15] Damien Imbs and Michel Raynal. 2012. Virtual world consistency: A condition for STM systems (with a versatile protocol with invisible read operations). *Theoretical Computer Science* 444 (2012), 113–127.
[16] Intel TBB [n. d.]. https://www.threadingbuildingblocks.org
[17] Idit Keidar and Dmitri Perelman. 2015. Multi-versioning in transactional memory. In *Transactional Memory. Foundations, Algorithms, Tools, and Applications.*
[18] Idit Keidar and Dmitri Perelman. 2015. On avoiding spare aborts in transactional memory. *ACM Transactions on Computer Systems* 57, 1 (July 2015), 261–285.
[19] Priyanka Kumar, Sathya Peri, and K. Vidyasankar. 2014. A timestamp based multi-version STM algorithm. In *International Conference on Distributed Computing and Networking.*
[20] Hsiang-Tsung Kung and John T Robinson. 1981. On optimistic methods for concurrency control. *ACM Transactions on Database Systems (TODS)* 6, 2 (1981), 213–226.
[21] Petr Kuznetsov and Sathya Peri. 2017. Non-interference and local correctness in transactional memory. *Theoretical Computer Science* 688 (2017), 103–116.
[22] David Lomet, Alan Fekete, Rui Wang, and Peter Ward. 2012. Multi-version concurrency via timestamp range conflict management. In *International Conference on Data Engineering.*
[23] Jeff Napper and Lorenzo Alvisi. 2005. *Lock-free Serializable Transactions.* Technical Report. University of Texas at Austin.
[24] Daniel Peng and Frank Dabek. 2010. Large-scale Incremental Processing Using Distributed Transactions and Notifications. In *Symposium on Operating Systems Design and Implementation.*
[25] Dmitri Perelman, Anton Byshevsky, Oleg Litmanovich, and Idit Keidar. 2011. SMV: Selective Multi-Versioning STM. In *International Symposium on Distributed Computing.*
[26] Dmitri Perelman, Rui Fan, and Idit Keidar. 2010. On maintaining multiple versions in STM. In *ACM Symposium on Principles of Distributed Computing.*
[27] Torvald Riegel, Pascal Felber, and Christof Fetzer. 2006. A Lazy Snapshot Algorithm with Eager Validation. In *International Symposium on Distributed Computing.*
[28] Yair Sovran, Russell Power, Marcos K. Aguilera, and Jinyang Li. 2011. Transactional storage for geo-replicated systems. In *ACM Symposium on Operating Systems Principles.*
[29] K Vidyasankar. 1987. Generalized theory of serializability. *Acta Informatica* 24, 1 (1987), 105–119.
[30] Gerhard Weikum and Gottfried Vossen. 2001. *Transactional information systems: theory, algorithms, and the practice of concurrency control and recovery.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.
[31] Yingjun Wu, Joy Arulraj, Jiexi Lin, Ran Xian, and Andrew Pavlo. 2017. An Empirical Evaluation of In-Memory Multi-Version Concurrency Control. *Proceedings of the VLDB Endowment* 10, 7 (March 2017), 781–792.
[32] Xiangyao Yu, Andrew Pavlo, Daniel Sanchez, and Srinivas Devadas. 2016. TicToc: Time Traveling Optimistic Concurrency Control. In *International Conference on Management of Data.*

A CORRECTNESS OF GENERIC MVTL ALGORITHM

Theorem 1. *The generic MVTL algorithm (Algorithms 1 and 2) ensures serializability.*

Proof. We denote by $T.committs$ the timestamp at which transaction T is serialized and commits (aborted transactions do not have a serialization timestamp). Each transaction has a unique serialization timestamp, as explained in §4.1. If a transaction T commits at a timestamp $T.committs$, then it holds write locks at $T.committs$ for all the data in its write set, and read locks from the largest timestamp smaller than $T.committs$ containing a committed value to $T.committs$ for all the data in its read set (Algorithm 1, line 13). We denote by $r_i[x_j]$ the fact that transaction T_i has read a version of object x written by transaction T_j (i.e., the read operation has returned $Values[x, T_j.committs]$). In addition, we denote by $w_k[x_k]$ the fact that transaction T_k has written a new version of object x (i.e., it has written a value to $Values[x, T_k.committs]$).

We assume the serialization order is given by the commit timestamp of the transaction. That is, if transaction T_1 creates version v_1 of object o, and transaction T_2 creates version v_2 of object o, we say $v_1 \ll v_2$ iff $T_1.committs < T_2.committs$.

Let H be a multiversion history over a set of transactions $\{T_0, \ldots, T_n\}$, and $C(H)$ the committed projection of this history. The committed projection of an operation history retains only the operations that belong to committed transactions. A multiversion serialization graph (MVSG) has the transactions $\{T_0, \ldots, T_n\} \in C(H)$ as vertices and edges (1) from T_i to T_j if T_j reads from T_i, and (2) for $r_k[x_j]$ and $w_i[x_i] \in C(H)$, if $x_i \ll x_j$, then the graph has an edge from T_i to T_j, otherwise it has an edge from T_k to T_i.

It has been shown [5] that if the multiversion serialization graph is acyclic, then a multiversion history is *one copy serializable*, that is, equivalent to a serial one version history.

Similarly to the proof of the original multiversion timestamp order algorithm, we show the MVSG resulting from MVTL is acyclic by showing that if an edge between T_i and T_j exists in the graph, $T_i.committs < T_j.committs$. We consider the types of edges that can appear in a multiversion serialization graph. The first type of edges are *reads-from edges*. In this case, transaction T_j reads a version written by transaction T_i. Function read-locks acquires locks for timestamps starting immediately after the timestamp containing the version whose value is returned (and, since it read-locks an interval of timestamps, does not lock timestamps equal or larger to later versions). Hence, the read can only be serialized at a timestamp higher than that at which the read version was created. Thus, $T_i.committs \leq T_j.committs$. The second type of edge appears if $r_k[x_j]$ and $w_i[x_i]$ are in H and $x_i \ll x_j$. In this case, an edge from T_i to T_j exists in the graph. By definition of $\ll$, $x_i \ll x_j$ iff $T_i.committs < T_j.committs$. Finally, the third type of edge appears if $r_k[x_j]$ and $w_i[x_i]$ are in H and $x_j \ll x_i$. In this case, an edge from T_k to T_i is created (this assumes $k \neq i$). Since $x_j \ll x_i$, we know that $T_j.committs < T_i.committs$. Given that T_k has performed a read of version x_j, T_k has necessarily applied read locks for each timestamp from $T_j.committs + 1$ to $T_k.committs$. A read lock can only be acquired if no write lock from another transaction is present. Similarly, a write lock on a timestamp cannot be acquired if a read lock from another transaction is present. Thus, $w_i[x_i]$ could not have occurred in the interval $[T_j.committs + 1, T_k.committs]$. And since we know $T_j.committs < T_i.committs$, $w_i[x_i]$ must have necessarily occurred after the interval. Thus, $T_k.committs < T_i.committs$. Given that all the edges in the graph are from transactions with lower serialization timestamps to transactions with higher serialization timestamps, a cycle cannot exist. Thus, H is one-copy serializable. □

B DETAILS OF THE PREFERENTIAL ALGORITHM

The MVTL-Pref algorithm is recalled in Algorithm 5. Each transaction is assigned a *preferential timestamp* and one or more alternative timestamps. The system tries to commit the transaction using first the preferential timestamp, but if that would abort the transaction, it tries the alternative timestamps. Transactions are serialized in the order of their commit timestamps.

More precisely, the algorithm is parameterized by a function $A(t)$ that takes the transaction's preferential timestamp and returns a non-empty *set* of alternative timestamps different from t. For example, $A(t) = \{t-10, t+10\}$ indicates that $t-10$ and $t+10$ are the alternative timestamps for a transaction with preferential timestamp t. The preferential timestamp itself comes from a clock, as in other timestamp-based protocols. Similarly, we assume that processes obtain unique timestamps (e.g., by appending the process id to each timestamp t) and that $A(t)$ also produces unique timestamps (e.g., by using the process id in t in each timestamp in $A(t)$).

When reading, the system acquires read-locks for a set that includes the preferential timestamp and as many other timestamps as possible. When committing, the system tries to write-lock on all objects in the write set and the preferential timestamp; if that is not possible, it tries each of the alternative timestamps.

We provide a more precise definition of the concept of a workload:

Definition 1. *A* workload *is a set of n transaction inputs, where each transaction input is a finite sequence of operation-timestamp pairs with increasing timestamps and an operation is either read(k), write(k, v) or tryCommit.*

We now show that under certain conditions on $A(t)$, MVTL-Pref is strictly better than MVTO+, in the sense that (a) if MVTO+ does not abort under a workload, then MVTL-Pref does not abort either, and (b) there are infinitely many workloads where MVTO+ aborts but MVTL does not. These results hold assuming that $A(t)$ contain only timestamps smaller than t, that is, the alternative timestamps are smaller than the preferential one.

Theorem 2. *Suppose that $\forall t' \in A(t), t' < t$. (a) If a workload W produces no abort under MVTO+, then W produces no abort under MVTL-Pref. (b) There are infinitely many workloads that produce no aborts under MVTL-Pref but produce aborts under MVTO+.*

Proof sketch. (a) Consider a workload W that does not abort under MVTO+. We prove that, for each transaction T in W, the execution of T under MVTO+ and MVTL-Pref will read- and write-lock exactly the same timestamps. The intuition here is that MVTL-Pref will choose the same timestamps as MVTO+ under workload W, because

Algorithm 5 The MVTL-Pref algorithm

```
1: function Initialization(tx)
2:     tx.PrefTS ← clock()
3:     tx.PossTS ← {tx.PrefTS} ∪ A(tx.PrefTS)
                                        ▷ possible timestamps for tx
4: function write-locks(tx, k) return ▷ lock write-set only on commit
5: function read-locks(tx, k)
6:     repeat
7:         tr ← max{t : t < tx.PrefTS and Values[k, t] ≠ ⊥}
                                        ▷ candidate value to read
8:         tmax ← max{t ∈ tx.PossTS :
                        no timestamps in [tr+1, tmax] are write frozen}
9:         for t ← tr+1 to tmax do   ▷ read-lock [tr+1, tx.TS] if possible
10:            try to acquire read-lock for tx on (k, t), waiting
                   if timestamp is write-locked but not frozen
11:            if found frozen write-lock then
                   release read-locks acquired above; break ▷ exit "for" loop
12:    until found no frozen locks in the for loop
13:    tx.PossTS ← tx.PossTS ∩ [tr, tmax]   ▷ update possible timestamps
14:    return tr
15: function commit-locks(tx)
16:    for t ∈ tx.PossTS do    ▷ Find a good timestamp. Loop order: first
                                   tx.PrefTS then arbitrary for PossTS
17:        gotlocks ← true
18:        for (k, tr) ∈ tx.writeset do
19:            try to write-lock for tx on (k, t), without waiting if a
                   timestamp is read-locked
20:            if write-lock not acquired then
21:                gotlocks ← false        ▷ this timestamp will not work
22:                release all write locks for tx
23:                break                   ▷ exit inner "for" loop
24:        if gotlocks then break ▷ found a timestamp for which we can
                                      get write locks; exit outer "for" loop
25:    if gotlocks then tx.TS ← t          ▷ found good timestamp
26:    else tx.TS ← ⊥                      ▷ no good timestamps
27: function commit-ts(T) return tx.TS
28: function commit-gc(tx) return false
```

W does not cause any aborts. More precisely, we can show that (i) whenever a read occurs, both MVTO+ and MVTL-Pref pick the same value to return for the read (the first non-$\perp$ value with a timestamp smaller than the preferential timestamp); because the preferential timestamp is higher than any of the timestamps in $A(t)$, the MVTL-Pref picks the preferential timestamp as *tmax* and therefore locks the same range as MVTO+. Moreover (ii), whenever a commit occurs, both MVTO+ and MVTL-Pref pick the same timestamp to lock. This is because MVTL-Pref picks the preferential timestamp, given that MVTO+ does not abort. From (i) and (ii), it is possible to show that MVTL-Pref executes in exactly the same way as MVTO+ under W. Therefore, MVTL-Pref does not abort any transactions under W.

(b) Pick three timestamps $t_1 < t_2 < t_3$ such that $\max A(t_2) < t_1$. These will be the timestamps for transactions T_1, T_2, T_3. Consider the following workload: $W_1(Y)\, C_1\, R_2(X)\, R_3(Y)\, C_3\, W_2(Y)\, C_2$. Under MVTO+, this workload aborts T_2 since the timestamp at which T_2 wants to write Y is between t_1 and t_3. However, under MVTL-Pref, T_2 commits because MVTL-Pref can pick the alternative timestamp $\max A(t_2)$ with which to commit T_2. It is easy to generalize this example to several transactions, and thus obtain infinitely many workloads where MVTO+ causes an abort but MVTL-Pref does not. □

C DETAILS OF THE PRIORITIZER ALGORITHM

The MVTL-Prio algorithm is given in Algorithm 6. Operations from transactions with priority try to lock timestamps up to $+\infty$: writes attempt to lock all timestamps, while reads lock from the latest observed write onwards; the transaction commits at the lowest timestamp that was locked for all its data items. In contrast, transactions with no priority behave identical to the MVTO+ algorithm: they read the clock at the beginning and try to serialize all operations at that point (thus only acquiring locks for timestamps lower than or equal to the clock value at the beginning of the transaction).

Theorem 3. *In the MVTL-Prio algorithm, transactions labeled critical are never aborted by transactions labeled normal.*

Proof sketch. Assume *maxts* is the maximum serialization timestamp of all completed or executing transactions with no priority. For any objects, transactions without priority will not prevent a transaction with priority from locking the interval $[maxts, +\infty]$, and thus committing at a timestamp at most *maxts*. Thus, transactions without priority cannot cause a transaction with priority to abort. □

D DETAILS OF THE ϵ-CLOCK ALGORITHM

The MVTL-ϵ-clock algorithm is recalled in Algorithm 7. It assumes that clocks are ϵ-synchronized and ensures that transactions never abort in serial executions.

Upon start, a transaction *tx* reads the clock, obtains a time t, and sets a local variable *tx.TS* to the interval $[t-\epsilon, t+\epsilon]$. This set has the timestamps that *tx* tries to lock as it executes. To write k, *tx* obtains a write-lock on as many timestamps in *tx.TS* as possible, waiting if any of the timestamps is read- or write-locked (but not frozen) by another transaction; if *tx* already holds a read-lock on a timestamp, it waits until it can upgrade it to a write-lock. Next, if *Tw* denotes the locks that *tx* actually manages to acquire, *tx* sets *tx.TS* to *Tw*.

To read k, *tx* selects the largest timestamp m in *tx.TS*, finds the largest timestamp $tr < m$ under which k has been written, and then tries to acquire a read-lock on $[tr+1, m]$ (if *tx* already has a write-lock then it does not need to acquire a read-lock), waiting if a timestamp is write-locked (but not frozen) by another transaction. *tx* may find a frozen write-lock if some other transaction commits after *tx* picked *tr*; In that case, *tx* picks *tr* again and retries. Then *tx* updates *tx.TS* to contain the locked timestamps.

To commit, *tx* picks the smallest locked timestamp and runs garbage collection before completing the commit.

Note that initially *tx.TS* contains the correct real-time *treal* when *tx* started. In a sequential execution, we show that *tx* picks a commit timestamp that is at most *treal*, and thus it releases the lock on higher timestamps. As a result, the next transaction in the sequence will always have its own real time in its *tx.TS*, so that does not abort.

Algorithm 6 The MVTL-Prio algorithm

```
1: function Initialization(tx)
2:     if tx.priority = false then tx.TS ← clock()
3: function write-locks(tx, k)
4:     if tx.priority = true then
5:         for t = +∞ downto 0 do                    ▷ write-lock all the possible timestamps
6:             try to acquire write-lock for tx on (k, t), waiting
               if a timestamp is read- or write-locked but not frozen
7: function read-locks(tx, k)
8:     if tx.priority = true then
9:         repeat
10:            tr ← max{t : t < tx.TS and Values[k, t] ≠ ⊥}
11:            for t = +∞ downto tr+1 do              ▷ read-lock interval [tr+1, +∞] if possible
12:                try to acquire read-lock for tx on (k, t), waiting
                   if timestamp is write-locked but not frozen
13:                if found frozen write-lock then release read-locks acquired above; break   ▷ exit the "for" loop
14:        until found no frozen locks in the for loop
15:    else
16:        repeat
17:            tr ← max{t : t < tx.TS and Values[k, t] ≠ ⊥}
18:            for t = tr+1 to tx.TS do               ▷ read-lock interval [tr+1, tx.TS] if possible
19:                try to acquire read-lock for tx on (k, t), waiting
                   if timestamp is write-locked but not frozen
20:                if found frozen write-lock then release read-locks acquired above; break   ▷ exit the "for" loop
21:        until found no frozen locks in the for loop
22:    return tr
23: function commit-locks(tx)
24:    if tx.priority = false then
25:        for (k, tr) ∈ tx.writeset do
26:            try to write-lock for tx on (k, tx.TS), without waiting if a timestamp is read-locked
27:            if write-lock not acquired then
28:                tx.TS = ∅ and release all write locks for tx;
29:                return ;
30: function commit-ts(T)
31:    if tx.priority = true then
32:        return min T
33:    else
34:        return tx.TS
35: function commit-gc(tx)
36:    if tx.priority = true then
37:        return true
38:    else
39:        return false
```

Algorithm 7 The MVTL-ϵ-clock algorithm

```
1: function Initialization(tx)
2:     now ← clock()
3:     tx.TS ← [now − ε, now + ε]
4: function write-locks(tx, k)
5:     try to write-locks for tx on (k, tx.TS), waiting
           if a timestamp is read- or write-locked but not frozen
6:     tx.TS ← write-locks that tx could acquire
7: function read-locks(tx, k)
8:     if tx.TS = ∅ then return ⊥
9:     m ← max tx.TS
10:    repeat
11:        tr ← max{t : t < m and Values[k, t] ≠ ⊥}
12:        for t = tr+1 to m do ▷ read-lock interval [tr+1, m] if possible
13:            try to acquire read-lock for tx on (k, t), waiting
                   if timestamp is write-locked but not frozen
14:            if found frozen write-lock then
                   release read-locks acquired above; break ▷ exit "for" loop
15:    until found no frozen locks in the for loop
16:    tx.TS ← tx.TS ∩ [tr+1, m]
17:    return tr
18: function commit-locks(tx) return
19: function commit-ts(T) return min T
20: function commit-gc(tx) return true
```

We now show that MVTL-ϵ-clock is not susceptible to serial aborts, which we define precisely as follows:

- *(Serial abort)* An algorithm is susceptible to serial aborts if it has a serial schedule that aborts some transaction.

Theorem 4. *The MVTL-ϵ-clock algorithm is not susceptible to serial aborts when clocks are ϵ-synchronized.*

Proof sketch. According to the ϵ-clock assumption, the local clock the transaction sees can diverge from the real time by at most ϵ. The first step a transaction takes when it starts is to read its local clock t. Assume t_{real_start} is the real time when local clock value t is read. Given that T starts with the interval $[t-\epsilon, t+\epsilon]$, it is guaranteed that $t_{real\_start} \in [t-\epsilon, t+\epsilon]$. At commit time, according to the ϵ-clock algorithm, a transaction commits with the smallest timestamp in its interval it was able to lock for all data items.

We show that if all transactions execute serially, each transaction will be able to commit, and that its commit point will not be larger than the real time at the beginning of the transaction. We prove this by induction:

Base case. Assume T_1 is the first transaction that executes serially in the system. The first point in its assigned interval $(t - \epsilon)$ will be at most equal to the real time at the start of the transaction. Given that no conflicting data exists in the system, this first transaction will be able to commit at this smallest timestamp in the interval.

Inductive step. Assume $n - 1$ transactions have executed serially, and have each committed at a timestamp that was at most equal to the real time at the respective start of the transaction. We now show the n-th serial transaction will also commit with a timestamp at most equal to the real time at which it started.

Given that transactions execute serially, we know that the n-th transaction begins only after the previous one has completed. According to the algorithm, a transaction completes only after it performs garbage collection. Therefore, assuming the transactions committed with timestamps at most equal to the real time when they started, after the first $n-1$ transactions commit, no lock is held for timestamps higher than the real time the $n-1$-th transaction started. As the transactions execute serially, the real time the n-th transaction starts is larger than the real time any of the previous transactions started, and thus higher than any lock held in the system (therefore, no conflict can arise for a serial transaction that tries to commit at this timestamp). As the interval assigned to transaction n is guaranteed to contain the real time at the transaction's start, the n-th transaction will be able to commit with a timestamp at most equal to the real time when it started.

□

If concurrent transactions start less than $2 * \epsilon$ time apart in real time, since operations always wait if timestamps are locked but not frozen, they may have to wait for each other's operations to complete. Therefore our algorithm intuitively behaves similarly to pessimistic concurrency control for these transactions. Thus, the trade-off with this algorithm is that deadlocks are possible, and the system requires a deadlock detection mechanism.

E DETAILS OF THE MVTL-TO ALGORITHM

The MVTL-TO algorithm is given in Algorithm 8. Each transaction chooses a serialization timestamp at the beginning, and attempts to serialize every operation at this timestamp. For reads, it finds the largest timestamp with a committed value smaller than its chosen serialization timestamp, applies read locks to every timestamp between these two, and returns the version's value. This is equivalent to reading the version with the largest timestamp smaller than the transaction timestamp and setting its *read-timestamp* in MVTO+. If a read encounters a timestamp that is write-locked, but not frozen, it waits. This wait is short: it stops when write locks that are not frozen are finally frozen.

For writes, the algorithm simply retains the values it wishes to write in its write set, without acquiring any locks. Only at commit time does the protocol try to lock the write set at the chosen serialization timestamp. If any read lock is encountered (frozen or not), the write lock is unsuccessful (since no garbage collection is performed). When a transaction fails to acquire a write lock, it releases all previously acquired write locks, and aborts. In case all the write locks are successfully acquired, they are then frozen and values are associated with the transaction's timestamp.

Theorem 5. *The MVTL-TO algorithm behaves as the MVTO+ algorithm.*

Proof sketch. Like MVTO+, MVTL-TO processes transactions such that they appear to execute in the order of their timestamp. The protocol provides all the properties of MVTO+, such as reads never aborting and only having read-write conflicts (given each process can choose unique timestamps, writes never conflict with other writes). □

Algorithm 8 The MVTL-TO algorithm

```
1:  function INITIALIZATION(tx)
2:      tx.TS ← clock()
3:  function WRITE-LOCKS(tx, k) return
4:  function READ-LOCKS(tx, k)
5:      repeat
6:          tr ← max{t : t < tx.TS and Values[k, t] ≠ ⊥}
7:          for t ← tr+1 to tx.TS do   ▷ read-lock interval [tr+1, tx.TS] if possible
8:              try to acquire read-lock for tx on (k, t), waiting
                    if timestamp is write-locked but not frozen
9:              if found frozen write-lock then release read-locks acquired above; break   ▷ exit the "for" loop
10:     until found no frozen locks in the for loop
11:     return tr
12: function COMMIT-LOCKS(tx)
13:     for (k, tr) ∈ tx.writeset do
14:         try to write-lock for tx on (k, tx.TS), without waiting if a timestamp is read-locked
15:         if write-lock not acquired then
16:             tx.TS = ∅ and release all write locks for tx
17:             return ;
18: function COMMIT-TS(T) return tx.TS
19: function COMMIT-GC(tx) return false
```

F DETAILS OF THE MVTL-PESSIMISTIC ALGORITHM

Briefly, the pessimistic concurrency control algorithm works as follows: as reads and writes are executed, they apply locks on the objects they access. At most one write can access any object at a point in time. If an object is locked for a write, no reads from other transactions can proceed concurrently. If a transaction cannot acquire a lock for an object, it waits until the lock is released. When all the locks are successfully acquired, the transaction performs its updates to the objects, and then unlocks.

This algorithm can be seen as a special case of MVTL with a specific policy, as shown in Algorithm 9. Basically, writes try to lock all possible timestamps, starting from $+\infty$ downwards, while reads also start from $+\infty$, and apply read locks to all timestamps down to the first timestamp where a write committed (whose value is also returned). If a transaction has successfully acquired locks for all its data, it will commit at the minimum timestamp that is locked for every data item (since such a timestamp always exists, the transaction will not abort—aborts can only potentially occur in case of deadlock). This timestamp will be equal to one greater than the largest timestamp of any read data, and is guaranteed to be less than $+\infty$. At the end of the transaction, the unneeded locks are released (including, in particular $+\infty$) and the next transaction can acquire locks for the concerned data items.

Proof sketch. Since both reads and writes first try to lock $+\infty$, it is guaranteed that at most one writer or multiple readers can have access to an object. Moreover, a transaction that has completed will never prevent other transactions from accessing any data object. □

Algorithm 9 The MVTL-Pessimistic algorithm

```
1: function WRITE-LOCKS(tx, k)
2:     for t = +∞ downto 0 do ▷ write-lock all the possible timestamps
3:         try to acquire write-lock for tx on (k, t), waiting
               if a timestamp is read- or write-locked but not frozen
4: function READ-LOCKS(tx, k)
5:     repeat
6:         tr ← max{t : t < m and Values[k, t] ≠ ⊥}
7:         for t = +∞ downto tr+1 do ▷ read-lock interval [tr+1, +∞] if possible
8:             try to acquire read-lock for tx on (k, t), waiting
                   if timestamp is write-locked but not frozen
9:             if found frozen write-lock then release read-locks acquired above; break ▷ exit the "for" loop
10:    until found no frozen locks in the for loop
11:    return tr
12: function COMMIT-LOCKS(tx) return
13: function COMMIT-TS(T) return min T
14: function COMMIT-GC(tx) return true
```

Algorithm 10 The MVTL-Ghostbuster algorithm

```
1: function INITIALIZATION(tx)
2:     tx.TS ← clock()
3: function WRITE-LOCKS(tx, k) return
4: function READ-LOCKS(tx, k)
5:     repeat
6:         tr ← max{t : t < tx.TS and Values[k, t] ≠ ⊥}
7:         for t = tr+1 to tx.TS do ▷ read-lock interval [tr+1, tx.TS] if possible
8:             try to acquire read-lock for tx on (k, t), waiting
                   if timestamp is write-locked but not frozen
9:             if found frozen write-lock then release read-locks acquired above; break ▷ exit the "for" loop
10:    until found no frozen locks in the for loop
11:    return tr
12: function COMMIT-LOCKS(tx)
13:    if tx.TS = ∅ then return
14:    for (k, tr) ∈ tx.writeset do
15:        try to write-lock for tx on (k, tx.TS), waiting
               if a timestamp is read- or write-locked but not frozen
16:        if write-lock not acquired then tx.TS = ∅ and release all write locks for tx;
17: function COMMIT-TS(T) return tx.TS
18: function COMMIT-GC(tx) return true
```

Theorem 6. *The MVTL-Pessimistic algorithm behaves as the pessimistic concurrency control algorithm.*

G DETAILS OF THE GHOSTBUSTER ALGORITHM

We now give the MVTL-Ghostbuster algorithm, which avoids ghost aborts. We start with a precise definition of ghost aborts. To do so, we first define the notion of an active conflict, which intuitively means a conflict with a transaction that is concurrently running. More precisely, given an execution of algorithm:

- *(Active conflict)* A transaction T_i has an active conflict if it has an operation o_i that conflicts with some operation o_j of another transaction T_j, where o_i is concurrent with T_j.
- *(Ghost abort)* An algorithm is susceptible to ghost aborts if it has a schedule where a transaction aborts but it has no active conflicts.[3]

The notion of ghost aborts is related to concepts discussed in previous work [15, 21, 29]. In particular, in the context of STMs, Kuznetsov and Peri [21] refer to a similar concept as transaction interference.

To avoid ghost aborts, an algorithm must ensure that each transaction that aborts has at least one operation with an active conflict.

The MVTL-Ghostbuster algorithm is shown in Algorithm 10. This algorithm is similar to MVTL-TO, which emulates MVTO, with the addition of garbage collection before a transaction commits or aborts.

Theorem 7. *The MVTL-Ghostbuster algorithm is not susceptible to ghost aborts.*

Proof sketch. MVTL-Ghostbuster chooses a timestamp at the beginning of the transaction, and it serializes transactions according to this timestamp. As in the MVTO algorithm, the only conflicts triggering aborts are read-write conflicts. If a transaction T_i aborts, it must have been because a write lock could not be acquired. This can only happen because a read lock already exists for $T_i.TS$ at the time of the write. If this is a ghost conflict, the lock must have been held by a transaction T_j that has finished its execution and aborted at the time of the conflict. But in real time, a transaction's commit method only finishes (with either an abort or commit result) after the GC function is called (in which function, if the transaction aborts, all its locks are removed). It is worth noting that in this algorithm, garbage collection is always performed. Hence, a transaction that aborts only holds any locks while it is executing (i.e., while it is an active transaction). Therefore, a write cannot encounter a conflict due to a transaction that already aborted, and thus no ghost conflicts can appear using this algorithm. □

[3] Ghost aborts are different from cascading aborts [30], which occur when a transaction reads uncommitted data.

H EXTENDING MVTL TO DISTRIBUTED SYSTEMS

For the distributed version of MVTL, we consider a standard distributed system model [6], with processes that communicate via message passing. The system is asynchronous: there are no bounds on the relative speed of processes or on communication. Processes have local clocks, with domain $\mathcal{T} = \{0, 1, \ldots\}$, which need not be synchronized. Unless explicitly stated otherwise, processes may exhibit crash-failures: they may stop executing unexpectedly. Where appropriate, we discuss other failure models as well. We assume the data is partitioned among multiple servers, and may or may not be replicated (we discuss both cases). Transactions are coordinated by the processes that want to execute them; we refer to such processes as clients or coordinators.

Algorithms 11 and 13 show the basic algorithm for the client and server respectively, while Algorithm 12 shows a generic policy. The policy is specified by the transaction coordinator, and it is applied by the server.

This generic algorithm leads to specific algorithms with high communication efficiency: only one round-trip to each object in the read set and two round-trips to each object in the write set. This efficiency is possible when the policy does not require garbage collection and its fault tolerance mechanism does not send messages when the coordinator is unsuspected (we discuss when this is viable in §H.1).

Algorithm 11 The generic distributed MVTL algorithm

```
1:  function BEGIN(tx)
2:      tx.readset ← ∅; tx.writeset ← ∅; tx.committs ← ⊥
3:  function WRITE(tx, k, v)                    ▷ write v to k in transaction tx
4:      status ← WRITE-LOCKS(tx, k, v)          ▷ write lock some subset of timestamps
5:      if status = abort then
6:          decision ← tx.commitment.tryAbort(); ▷ decision must be abort in this case
7:          mark tx as aborted
8:          return
9:      add (k, v) to tx.writeset               ▷ remember key and value we wrote
10: function READ(tx, k)                        ▷ read k in transaction tx
11:     (tr, V) ← READ-LOCKS(tx, k)             ▷ read lock some interval [tr+1, . . .] with Values[k, tr] ≠ ⊥
12:     if tr = ⊥ then return ⊥                 ▷ read failed
13:     add (k, tr) to tx.readset               ▷ remember key and version we read
14:     return V                                ▷ return committed value
15: function COMMIT(tx)                         ▷ try to commit transaction tx
16:     COMMIT-LOCKS(tx)                        ▷ locks to acquire at commit time
17:     T ← {t : ∀k ∈ tx.readset.keys, tx has a lock on (k, t) and ▷ try to find a locked timestamp for tx
              ∀k ∈ tx.writeset.keys, tx has a write-lock on (k, t)}
18:     if T = ∅ then
19:         decision ← tx.commitment.tryAbort(); ▷ decision must be abort in this case
20:         mark tx as aborted
21:     else
22:         tx.committs ← COMMIT-TS(T)          ▷ pick some timestamp in T
23:         decision ← tx.commitment.tryCommit(tx.committs);
24:         if decision = abort then
25:             mark tx as aborted
26:         else
27:             for (k, v) ∈ tx.writeset do
28:                 send(server(k), freeze-write-lock, k, tx.committs)   ▷ freeze locks
29:     if COMMIT-GC(tx) then GC(tx)            ▷ invoke gc or not
30: function GC(tx)                             ▷ garbage collect locks of tx after it ended
31:     if tx committed then
32:         for (k, tr) ∈ tx.readset do
33:             send(server(k), freeze-read-locks, k, [tr+1, tx.committs])
34:     send messages to release all unfrozen read- and write-locks for tx
```

Relative to the centralized MVTL algorithm, the main technical challenge addressed by the distributed MVTL algorithm is handling failures. A transaction coordinator failure may leave write locks in an unfrozen state indefinitely, causing other transactions to block

Algorithm 12 Client policy for the generic distributed MVTL algorithm

```
1:  function WRITE-LOCKS(tx, k, v)
2:      send(server(k), (tx, write-locks, k, v, T)), for some set T
3:      wait_message(server(k), status, T′)   ▷ T′ subset of T for which write locks acquired
4:      return status
5:  function READ-LOCKS(tx, k)                ▷ returns a timestamp or ⊥
6:      send(server(k), (tx, read-lock, k, T, criteria)), for some set T
7:      wait_message(server(k), tr, te, V)    ▷ [tr + 1, te] read locked if tr ≠ ⊥, V read value
8:      either return (tr,V) or return (⊥, bot)
9:  function COMMIT-LOCKS(tx)
10:     acquire read- or write-locks for tx on some keys and timestamps as above
11: function COMMIT-TS(T) return some t ∈ T
12: function COMMIT-GC(tx) either return true or return false
```

Algorithm 13 The server

```
1:  function RECEIVE-WRITE-LOCK-MESSAGE(tx, k, v, T)
2:      acquire write-locks for tx on (k, T′) for T′ ∈ T in which acquiring locks is possible
3:      tx.pending_value(k) ← v;              ▷ remember v as new value
4:      send(client(tx), write-locks-acquired, T′)
5:  function RECEIVE-READ-LOCK-MESSAGE(tx, k, T, criteria)
6:      acquire read-locks for tx on (k, I) for I = [tr+1, te] where te ∈ T chosen according to criteria and Values[k, tr] ≠ ⊥
7:      send(client(tx), read-locks-acquired, tr, te or ⊥, Values[k, tr] or ⊥)
8:  function RECEIVE-FREEZE-WRITE-LOCK-MESSAGE(tx, k, t)
9:      decision ← tx.commitment.tryCommit(t)
10:     if decision = abort then
11:         release tx′s write locks
12:         return
13:     freeze write-lock for tx on (k, t)    ▷ freeze locks
14:     Values[k, t] ← tx.pending_value(k)    ▷ expose committed value
15:     send(client(tx), write-locks-frozen, k)
16: function RECEIVE-FREEZE-READ-LOCK-MESSAGE(tx, k, [start, commit])
17:     freeze read-locks for tx on k for [start, commit]
18:     send(client(tx), read-locks-frozen, k)
19: function WRITE-LOCK-TIMEOUT(tx)
20:     decision ← tx.commitment.tryAbort()
21:     if decision = "commit @ t" then
22:         freeze write-lock for tx on (k, t)    ▷ freeze locks
23:         Values[k, t] ← tx.pending_value(k)    ▷ expose committed value
24:         send(client(tx), write-locks-frozen, k)
25:     else                                  ▷ decision = abort
26:         release tx′s write locks
```

forever. A server failure similarly causes either indefinite waiting from transaction coordinators or failure of all transactions accessing the failed server.

The solution to both types of failure is simple: we associate a *commitment* object with each transaction, to ensure that everyone agrees on whether the transaction committed or aborted. Technically, the commitment object solves consensus: it ensures that (1) no two processes obtain different decisions, (2) the only possible decisions are *abort* or *commit*(t) where t is a timestamp, (3) if the decision is d, some participant proposed d, (4) each correct process eventually decides, and decides only once.

After a coordinator has acquired all the necessary locks and has found a commit timestamp, it proposes the *commit* outcome with the associated timestamp to the commitment object of the transaction. If the decision is to commit, the coordinator then proceeds to inform the servers in the write set of the commit timestamp, without waiting for replies, allowing them to freeze the write locks associated with this transaction (we note that the protocol would be correct even without this step; we include it for performance). When the servers receive a request to freeze the locks of a transaction, they also propose *commit* with the received timestamp from the coordinator. This is because from the point of view of a server, when it has received the serialization timestamp of a transaction, the transaction is committed. However, if a server has held unfrozen write locks for a certain amount of time without receiving the freeze message from the coordinator, it will assume the coordinator has failed and it will propose an *abort* outcome to the commitment object. If the decision is to commit, the server will receive a timestamp along with the decision and will be able to simply freeze its write locks at that timestamp and consider the transaction committed. The server can make this assumption because a commit decision is only possible if someone proposed *commit*, and a commit proposal only happens after the coordinator has performed all its updates and has found a commit timestamp, or after the coordinator has already informed a server of the commit timestamp. In both these instances the transaction can be committed. In the eventuality of an *abort* decision from the commitment object, a server releases all the write locks associated with that transaction and considers it aborted.

H.1 Commitment object implementations

This general mechanism used in our protocol allows various commitment object implementations, depending on the failure model we assume. If the coordinator or any minority of servers may fail, a Paxos-like consensus protocol could be used, with all the servers in the system as participants. This is because no server knows the write set of transactions, which can change dynamically with the execution.

However, in practice, storage servers are often replicated and their failures are masked, to provide both availability and durability of data. In this case, we can consider the storage server as a logical entity that does not fail, and consider only failures of the coordinator. By doing so, we can obtain an efficient implementation of commitment, one that requires little communication in the common failure-free case. We do so by implementing the commitment object using Terminating Reliable Broadcast (TRB) [13], as we now explain.

Essentially, the coordinator designates a single server per transaction as the *decision point*. This server can be, for example, the first server accessed by a write operation. Consensus on the outcome of the reliable broadcast (i.e., whether the source has delivered or has crashed) is achieved on this decision server. Servers accessed on subsequent writes will then be informed of the decision point of the transaction. When the coordinator proposes a value to the commitment object, it needs to inform the decision point of this proposal and wait for its decision. When proposing a commit, the message can also act as a *freeze-write-locks* message to the decision server, which is only applied if the decision is to commit. If the coordinator has proposed *abort*, no other outcome is possible (since without receiving a *commit* message from the coordinator, servers themselves can only propose *abort*). However, in case of a commit proposal, the outcome may be that of *abort*: when write locks have been acquired for a certain amount of time, but have not been frozen, the servers suspect the coordinator of having failed, and thus propose *abort*. The *abort* proposal from a server is similar to that of the coordinator: the decision point is contacted, and its decision is followed. If the coordinator's commit proposal has been executed at the decision point earlier than any *abort* proposal, the decision will be to commit, and the decision server sends the commit timestamp along with the decision. A server proposes commit only once it has received the *freeze-write-locks* message. But this message is only sent by the coordinator if the decision has been to commit. Hence, the commit proposal that a server does when freezing write locks can be executed entirely locally. If the server has not been informed of a transaction abort, it simply stores the *commit* decision locally. Thus, in the common, failure-free case, the coordinator does not need to exchange extra messages to be able to ensure fault tolerance.

H.2 Correctness

We start by proving the following lemma concerning the outcome of a transaction:

Lemma 1. *In Algorithms 11 and 13, with the generic policy in Algorithm 12, if a participant considers a transaction as committed, no other participant considers it as aborted.*

Proof. The *commit* object associated with each transaction provides the standard properties of uniform consensus:

- *(Termination.)* Every correct process eventually decides some value.
- *(Validity.)* If a process decides v, then v was proposed by some process (and, as previously mentioned, v can only be *abort* or *commit*).
- *(Integrity.)* No process decides twice.
- *(Agreement.)* No two processes decide differently.

Each process, be it the coordinator or a server, uses the commit object in order to obtain the decision as to whether the transaction should be committed or aborted. Before this, it makes no assumptions about the state of a transaction. At commit time, the coordinator proposes *abort* if no serialization point was found, and commit otherwise. A server that times out proposes *abort*, and a server that

receives a freeze write lock message (essentially a commit message) proposes *commit*. By the agreement property, all the processes involved with a particular transaction obtain the same outcome of the transaction.

□

Using Lemma 1, we now prove the following theorem:

Theorem 8. *Algorithms 11 and 13 with the generic policy in Algorithm 12 ensures serializability.*

The proof is largely similar to the centralized version, but we recall it here for completeness.

Proof. We denote by $T.committs$ the timestamp at which transaction T is serialized and commits (aborted transactions do not have a serialization timestamp). Each transaction has a unique serialization timestamp, as explained in §4.1. If a transaction T commits at a timestamp $T.committs$, then it holds write locks at $T.committs$ for all the data in its write set, and read locks from the largest timestamp smaller than $T.committs$ containing a committed value to $T.committs$ for all the data in its read set (Algorithm 11, line 17). By Lemma 1, if the coordinator considers a transaction to be committed, no server can consider it to be aborted, and thus the locks of the transaction must still be held. We denote by $r_i[x_j]$ the fact that transaction T_i has read a version of object x written by transaction T_j (i.e., the read operation has returned $Values[x, T_j.committs]$). In addition, we denote by $w_k[x_k]$ the fact that transaction T_k has written a new version of object x (i.e., it has written a value to $Values[x, T_k.committs]$).

We assume the serialization order is given by the commit timestamp of the transaction. That is, if transaction T_1 creates version v_1 of object o, and transaction T_2 creates version v_2 of object o, we say $v_1 \ll v_2$ iff $T_1.committs < T_2.committs$.

Let H be a multiversion history over a set of transactions $\{T_0, \ldots, T_n\}$, and $C(H)$ the committed projection of this history. The committed projection of an operation history retains only the operations that belong to committed transactions. A multiversion serialization graph (MVSG) has the transactions $\{T_0, \ldots, T_n\} \in C(H)$ as vertices and edges (1) from T_i to T_j if T_j reads from T_i, and (2) for $r_k[x_j]$ and $w_i[x_i] \in C(H)$, if $x_i \ll x_j$, then the graph has an edge from T_i to T_j, otherwise it has an edge from T_k to T_i.

It has been shown [5] that if the multiversion serialization graph is acyclic, then a multiversion history is *one copy serializable*, that is, equivalent to a serial one version history.

Similarly to the proof of the original multiversion timestamp order Algorithm, we show the MVSG resulting from MVTL is acyclic by showing that if an edge between T_i and T_j exists in the graph, $T_i.committs < T_j.committs$. We consider the types of edges that can appear in a multiversion serialization graph. The first type of edges are *reads-from edges*. In this case, transaction T_j reads a version written by transaction T_i. Function read-locks acquires locks for timestamps starting immediately after the timestamp containing the version whose value is returned (and, since it read-locks an interval of timestamps, does not lock timestamps equal or larger to later versions). Hence, the read can only be serialized at a timestamp higher than that at which the read version was created. Thus, $T_i.committs \leq T_j.committs$. The second type of edge appears if $r_k[x_j]$ and $w_i[x_i]$ are in H and $x_i \ll x_j$. In this case, an edge from T_i to T_j exists in the graph. By definition of $\ll$, $x_i \ll x_j$ iff $T_i.committs < T_j.committs$. Finally, the third type of edge appears if $r_k[x_j]$ and $w_i[x_i]$ are in H and $x_j \ll x_i$. In this case, an edge from T_k to T_i is created (this assumes $k \neq i$). Since $x_j \ll x_i$, we know that $T_j.committs < T_i.committs$. Given that T_k has performed a read of version x_j, T_k has necessarily applied read locks for each timestamp from $T_j.committs + 1$ to $T_k.committs$. A read lock can only be acquired if no write lock from another transaction is present. Similarly, a write lock on a timestamp cannot be acquired if a read lock from another transaction is present. Thus, $w_i[x_i]$ could not have occurred in the interval $[T_j.committs + 1, T_k.committs]$. And since we know $T_j.committs < T_i.committs$, $w_i[x_i]$ must have necessarily occurred after the interval. Thus, $T_k.committs < T_i.committs$. Given that all the edges in the graph are from transactions with lower serialization timestamps to transactions with higher serialization timestamps, a cycle cannot exist. Thus, H is one-copy serializable. □

We now focus on the liveness guarantees of the protocol.

Lemma 2. *If the coordinator does not propose commit for a transaction, no server does either.*

Proof sketch. Servers only propose commit when receiving a freeze write locks request from the coordinator. However, the coordinator only sends these messages once it has proposed to commit the transaction and has received a positive decision. Hence, the servers cannot propose a commit before the coordinator does.

□

Lemma 3. *If a coordinator that has obtained write locks but has not committed fails, it is eventually suspected by every correct server that holds unfrozen write locks for the coordinator's ongoing transaction.*

Proof sketch. The proof is straight-forward, as every server holding unfrozen write locks suspects the coordinator after a certain (finite) amount of time has passed since the locks were acquired (and the *Write-Lock-Timeout* function is called).

□

Lemma 4. *If a coordinator fails before committing a transaction, its write locks are eventually released and the transaction aborted on the correct servers.*

Proof sketch. A transaction is effectively committed when the coordinator proposes *commit* to the commitment object corresponding to the transaction, and obtains the same decision. If a coordinator fails before proposing *commit*, according to Lemma 2, no-one proposes *commit* for its transaction. Therefore, a *commit* decision cannot be reached (according to the Validity property of the commitment object). According to Lemma 3, every server currently holding unfrozen write locks for the coordinator's ongoing transaction at the time of failure suspects it to have failed. In Algorithm 13, when a server suspects a coordinator (when a time-out for the unfrozen write-locks occurs), it proposes *abort*. Thus, since *commit* cannot be proposed, and every server holding write locks must propose *abort*, the only decision that can be reached is to abort.

□

We now prove the following theorem:

Theorem 9. *No transaction initiated by a correct coordinator is indefinitely delayed by a failed coordinator.*

Proof sketch. Indefinite delays can happen in one scenario: when unfrozen write locks are held for an object. A read at a higher timestamp (no matter how high) whose result would depend on whether or not a new version of the object is created at the timestamps that are currently write-locked but not frozen has no choice but to wait. Other operations may be affected by the ongoing transaction of a failed coordinator, but this does not result in waiting, but rather in aborting, and potentially retrying at a higher timestamp, where the two transactions would not interfere. According to Lemma 4, either a coordinator is correct, and thus eventually commits or aborts its transaction, or its write locks are eventually released. Therefore, it cannot be the case that a transaction initiated by a correct coordinator is delayed indefinitely by a failed coordinator. □

Theorem 10. *Unless at least one server suspects the coordinator to have failed, a transaction that has chosen a serialization timestamp eventually commits.*

Proof sketch. Servers only propose *abort* when suspecting the coordinator to have failed (i.e., unfrozen write locks have been held for too long). Thus, if the coordinator is not suspected, no server proposes *abort*. Thus, the only proposal servers can make in this scenario is to commit. Additionally, if the coordinator has found a serialization timestamp for the transaction, then it must propose *commit*. Thus, since no-one in the system proposes *abort*, and the coordinator must propose *commit*, the final decision of the commitment object must be to commit.

□